GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2023-2024

UNIVERSIDAD
DE GRANADA

INTELIGENCIA ARTIFICIAL

Memoria de la Práctica 3 -Búsqueda con Adversario (Juegos). El Parchís (BOOM!)

Ángel Rodríguez Faya

07/06/2024

ÍNDICE

1.-Introducción.

En esta práctica, la tercera de la asignatura de Inteligencia Artificial, trata sobre el diseño e implementación de técnicas de búsqueda con adversario en un entorno de juegos, en esta ocasión se trata de un Parchís determinista.

2.-Análisis del problema.

La readaptación del juego del Parchís consiste en sustituir el comportamiento aleatorio de tirar un dado por la elección del dado entre los dados disponibles. El conjunto de dados disponible serán 6 valores de un dado (1, 2, 4, 5, 6 y especial). Cada vez que se utilice uno de los dados ese valor se gastará, teniendo que elegir en el siguiente turno un dado diferente. Eventualmente, a los 6 valores del dado se añadirán valores especiales como 10 o 20 para ser utilizados en el momento. En este caso siempre habrá 2 jugadores que jugarán con dos colores alternos. Cada vez que le toque a un jugador, con el dado que elija sacar podrá mover una ficha de cualquiera de sus dos colores. Aunque cada jugador controle a dos colores, estos colores podrán atacarse entre sí. Ganará la partida el primer jugador que consiga meter todas las fichas de cualquiera de sus colores, independientemente de dónde estén el resto de sus fichas.

3.-Algoritmo de Poda Alpha-Beta.

```
double AIPlayer::podaAlphaBeta(const Parchis &actual, int jugador, color &c_piece, int &id_piece, int &dice, int profundidad,
    int profundidad_maxima, double alpha, double beta, double (*fun_heuristica)(const Parchis &, int)) const
{
    // CONDICION DE PARADA
    // Si la profundidad es la máxima o ha terminado la partida
    if(profundidad == profundidad_maxima or actual.gameOver()){

        // Devuelve la valoración del estado actual.
        return fun_heuristica(actual, jugador);
    }

    ParchisBros hijos = actual.getChildren();

    for(ParchisBros::Iterator it = hijos.begin(); it != hijos.end(); ++it){

        if(jugador == actual.getCurrentPlayerId()){ // Es un nodo MAX

            double valor = podaAlphaBeta(*it, jugador, c_piece, id_piece, dice,
            profundidad + 1, profundidad_maxima, alpha, beta, fun_heuristica);

            if(valor > alpha){

                alpha = valor;

                if(profundidad == 0){ // Compruebo si estamos en el nodo raiz
                    c_piece = it.getMovedColor();
                    id_piece = it.getMovedPieceId();
                    dice = it.getMovedDiceValue();
                }
            }
```

```
            // Verifico si se cumple la condición de poda.
            if(alpha >= beta){
                break;
            }

        }

        else{ // Es un nodo MIN

            double valor = podaAlphaBeta(*it, jugador, c_piece, id_piece, dice,
            profundidad + 1, profundidad_maxima, alpha, beta, fun_heuristica);

            if(valor < beta){

                beta = valor;
            }

            // Verifico si se cumple la condición de poda.
            if(alpha >= beta){
                break;
            }
        }
    }

    // Devuelvo el valor de alpha o beta, según si soy MAX o MIN.
    if(jugador == actual.getCurrentPlayerId()){
        return alpha;
    }
    else{
        return beta;
    }
}
```

4.-Mis heurísticas.

En primer lugar me gustaría destacar que, a pesar de que en este documento y en el código solamente muestre 4 heurísticas, he desarrollado más conforme he ido haciendo la práctica, pero no eran lo suficientemente competitivas ya que era muy fácil ganarles y realizaban movimientos fuera de lugar o que no eran los más apropiados para ganar la partida. A medida que avanzamos (probamos las heurísticas) vemos que van batiendo a nuestros rivales, los ninjas.

La mejor heurística es miHeuristica3(...), es por eso que la dejaré para el final de este apartado.

4.1.- miHeuristica1().

La cabecera de esta función se encuentra en AIPlayer.h, y es:

```
static double miHeuristica1(const Parchis &estado, int jugador);
```

y es implementada en el archivo AIPlayer.cpp.

En el método think, la podemos encontrar con el **id=2.**

```
case 2:
    valor = podaAlphaBeta(*actual, jugador, c_piece, id_piece, dice, 0, PROFUNDIDAD_ALFABETA, alpha, beta, miHeuristica1);
    break;
```

Esta heurística es la primera que realizé y como se podrá observar más adelante es una mejora de *ValoracionTest*, la heurística proporcionada por los profesores de la asignatura a modo de ejemplo. miHeuristica1() toma como parámetros el estado actual del tablero y el id del jugador actual. Lo primero que hace es obtener el id del jugador y de su oponente y comprobar si hay un ganador en la partida, devolviendo gana (+∞) o si pierde (-∞), **esta parte es heredada de *ValoracionTest* y estará presente en todas las heurísticas que he realizado.**

En caso de que todavía no hayamos ganado o perdido la partida, pasamos a calcular las puntuaciones del jugador y del oponente (estas quedarán reflejadas cada una en una variable). Para ello, empezamos calculando la puntuación del jugador.

Vamos a recorrer todas las fichas del jugador. Empezamos escogiendo los colores de mi jugador y después, por cada color, vamos recorriendo sus fichas una a una. A la puntuación del jugador le vamos a ir sumando por cada ficha si es una casilla segura, meta o pasillo de la meta y restando si es casa.

A continuación, **a la puntuación del jugador se le sumaría lo que nos devuelva la función *puntuacionPorDadoEspecial (const int valorPowerBar)*** que es una función creada por mí y que es utilizada en las mis heurísticas 1 y 2. Esta función se le pasa como parámetro el valor de la barra de energía de nuestro jugador para calcular una puntuación que será positiva o negativa dependiendo de si el objeto que tenemos en este momento nos beneficia o nos perjudica si utilizáramos el dado especial en este momento.

Una vez calculada la puntuación del jugador, pasamos a calcular la **puntuación del oponente,** que sería igual que la del jugador pero cambiando los signos de las operaciones, es decir, lo que al jugador le suma al oponente le resta y viceversa.

A continuación, **a la puntuación del oponente se le restaría lo que nos devuelva la función *puntuacionPorDadoEspecial(const int valorPowerBar).***

Al final, sólo quedaría devolver **puntuación_jugador - puntuacion_oponente.**

Ventajas de esta heurística:

+ Tiene en cuenta las casillas de casa y del pasillo de la meta, además de las casillas seguras y las de meta. Además, también tiene en cuenta el valor de la barra de energía.

Desventajas de esta heurística:

- **Las fichas de un mismo jugador se comen entre ellas.** No se ha tenido en cuenta esta situación a la hora de realizar la heurística, por lo que las fichas de un color del jugador se podrán a comerse las del otro color y viceversa, lo que hace imposible que gane la partida.

- **La puntuación del oponente se calcula de forma contraria a la del jugador.** Por desconocimiento mío y por la inexperiencia acerca de esta práctica, pensé en que la puntuación del oponente debía de calcularse de forma contraria a la del jugador. Esto no debería ser así, sino que se debería de calcular de igual manera. Este error fue subsanado en las siguientes heurísticas.

- **No es capaz de derrotar a algún ninja.**

4.2.- miHeuristica2().

La cabecera de esta función se encuentra en AIPlayer.h, y es:

```
static double miHeuristica2(const Parchis &estado, int jugador);
```

y es implementada en el archivo AIPlayer.cpp.

En el método think, la podemos encontrar con el **id=3.**

```
case 3:
    valor = podaAlphaBeta(*actual, jugador, c_piece, id_piece, dice, 0, PROFUNDIDAD_ALFABETA, alpha, beta, miHeuristica2);
    break;
```

Esta heurística implementa varios cambios con respecto en la anterior, como lo son:

- Se han reducido el valor de las fichas que están en la casa, en casilla segura, en la meta o en el pasillo de está.
- Se han añadido nuevas funcionalidades como:
    - Control sobre las fichas comidas, tanto las del jugador actual como del oponente.
    - Se valora positivamente la distancia a la meta, cuanto menor sea esta, mayor será el valor que se sume a la puntuación actual.
    - Se ha tenido en cuenta los valores del dado que me pueden beneficiar (Movimiento ultra-rápido, movimiento bala y la estrella), en los que sumarán a la puntuación del jugador.
    - Ahora se resta a la puntuación obtenida el número de piezas que hay en la casa multiplicado por 10.

Por lo demás sigue siendo igual a la heurística anterior, primero se calcula la puntuación del jugador, con lo que ya teníamos de antes, teniendo en cuenta lo que hemos modificado o añadido como he descrito anteriormente, se sigue llamando a la función puntuaciónPorDadoEspecial(...) etc.

Ventajas de esta heurística:

+ **Mejor que miHeuristica1().** Corrige los puntos débiles de la anterior heurística, siendo más competente y teniendo en cuenta más aspectos.

Desventajas de esta heurística:

- **Todavía no es vencedora.** A pesar de ganar al ninja 1 y 2 (solamente como jugador 1), no es capaz de ganar las otras 4 batallas que se disputan.

4.3.- miHeuristica3().

Llegamos a la heurística que **he elegido para que sea evaluada** mi solución propuesta para esta práctica.

Como vengo haciendo en las heurísticas anteriores, su cabecera es:

```
static double miSub_Heuristica3_1(const Parchis &estado, int jugador);
```

En el método think, la podemos encontrar con el **id=1.**

```
case 1:
    valor = podaAlphaBeta(*actual, jugador, c_piece, id_piece, dice, 0, PROFUNDIDAD_ALFABETA, alpha, beta, miHeuristica3);
    break;
```

En esta heurística he introducido un cambio, ya que verdaderamente el propósito de esta función es llamar a otra sub-heurística, una vez con el id del jugador y otra vez con el id del oponente y hacer la diferencia entre ellas. Esta subHeurística de la que hablo es la que he llamado en la clase AIPlayer como **miSubHeuristica3_1.**
El motivo de modularizar más las heurísticas no es otro que el de intentar reducir las líneas de código, ya que para calcular las puntuaciones (tanto del jugador como del oponente) se hace de la misma forma.

4.3.1- miSub_Heuristica3.1().

Esta es la mejor heurística que he conseguido hacer. Es parecida a la heurística 2, pero incorpora cambios como:

- Se han vuelto a reajustar los valores según las casillas (final_queue, home, etc).
- Se han reajustado también los valores que tienen el dadoEspecial, es decir, se ha suprimido la función puntuaciónPorDadoEspecial(...), su código se ha incluido en el de la subHeuristica3_1 y además se han ajustado los valores que sumaban/restaban a la puntuación total.

Ventajas:
+ **Vence a los ninjas.** Gana a los ninjas con facilidad y se muestra superior.
+ **Es rápida.** Junto al algoritmo de poda alpha-beta, no tarda mucho en tomar una decisión, como si pasaba en otras heurísticas.
+ **Es completa.** Tiene en cuenta muchos aspectos y situaciones, por lo que no se escapará ningún detalle.
+ **Hace una buena gestión del Dado Especial.** En las partidas que he realizado con esta heurística, nunca ha cogido el dado especial cuando había algún objeto que le podría repercutir negativamente en el resultado.

Desventajas:
- **Le cuesta sacar las fichas de la casa.** La única desventaja que le he visto es que a pesar de tener el dado 5 disponible, no sacaba la ficha de la casa. Intenté implementarlo pero no salió bien. Esta es la única desventaja que le veo a mi heurística.

5.-Conclusión.

Me habría gustado que nos hubieran permitido un par de hojas más en la memoria, ya que me he visto un poco ajustado para intentar expresar todas las ideas de mis heurísticas junto con sus puntos fuertes y débiles. Además, me habría gustado añadir códigos, al menos de la heurística 3. Pienso que hubiese quedado más claro ya que solamente mirando la memoria se podría haber expresado todo, sin necesidad de abrir un editor de código. También me gustaría señalar **que hay una miSub_heurística3_2(),** en la que intenté salirme un poco de lo que venía haciendo anteriormente y que **se enfocará en las fichas del color del jugador que mejor llevasen la partida, es decir, que tuviese más fichas en casa o más cerca.**

De todas formas, he intentado explicarme con la mayor claridad y brevedad posible. En el código delimitaré bien cada cosa y documentaré adecuadamente los métodos.

Me he divertido realizando la práctica y la idea de este Parchís me ha gustado, tiene un software bastante bueno y es fácil de usar y sobre todo útil. Pero lo que más quiero destacar es el aprendizaje que he adquirido sobre este tema de la asignatura, aprendiendo a hacer heurísticas, implementar el algoritmo de poda, etc.

En general, estoy muy contento con esta práctica. Enhorabuena a los profesores y gracias.